Software Complexity Measures of Bubble, Heap and Shell Sorting Algorithms Using Node Representation

Ayodele, Oluwakemi Sade[1]; Owoeye, Folusho Olayinka[1], Ezekiel, Rock Adeiza[1]; & Ajayi, Ebenezer Akinyemi[2]
[1]Department of Computer Science, Kogi State Polytechnic, Lokoja, Nigeria
[2]Department of Computer Science, Kebbi State Polytechnic, Dakingari. Nigeria
E-mail: Kemtemmy2009@gmail.com

ABSTRACT

One of the central problems in software engineering is the inherent complexity. Software metrics can provide us with information regarding the quality of software. In this thesis Software Complexity was measured. One method to determine the software complexity proposed in literature by Torn et al (1999) (structural complexity model called effort-length complexity metrics (ELC)) was used and the method was validated using a set of sorting algorithms (Bubble, Heap and Shell) written in C, Pascal, Java and Visual BASIC in its node representation. Results which are intuitively correct were obtained; higher values were obtained for average structural complexity and total complexity for the programs which was observed to be more complex than the others, not only in terms of length of program but also in terms of the contained structures. The results obtained by using the model agreed with those obtained when mathematical approach is used for the studied algorithms. Thus, using the model we obtain the results that previous Computer Scientists have obtained using different measures.

Keywords: Software engineering, complexity, structural complexity, node representation, structure.

CISDI Journal Reference Format
Ayodele, O.S.; Owoeye, F.O., Ezekiel, R.A. & Ajayi, E.A. (2023): Software Complexity Measures of Bubble, Heap and Shell Sorting Algorithms Using Node Representation. Computing, Information Systems, Development Informatics & Allied Research Journal. Vol 13 No 4, Pp 1-22. Available online at https://www.isteams.net/cisdijournal. dx.doi.org/10.22624/AIMS/CISDI/V14N4P1

1. BACKGROUND OF THE STUDY

In recent years, software complexity measurement has been the subject of considerable research. The scope of studying it was to control the levels of the external attributes of software via internal attributes, like complexity is. The most well-known internal attribute is software length and another very important attribute is complexity. While the case of length is a quite well-defined consensus about the ways the length should be measured, in the case of complexity there is still a lot of confusion. Antinyan et al. (2017).

Wikipedia Contributors (2019). Many software complexity metrics have been proposed and used to measure different software properties. McCabe proposes a graph-theoretic cyclomatic complexity (McCabe metric). GeeksforGeeks (2017). Halstead proposes measures to predict understanding effort based on grammatical complexity of code modules. Those measurements represent the properties they purpose to quantify. The software complexity is comprised of mainly two types, internal complexity and external complexity, sometimes be called micro complexity and macro complexity. The external complexity is contributed by inter-relationships among the software modules, whereas internal complexity is contributed by software module, program code. (Banker et al., 1989) Team (2020). It is not wrong to say that there is a relationship between complexity and the length of the program. But, all authors agree that when measuring complexity one should take into account something different from length and length at the same time.

This approach was followed in Torn A., Andersson T. and Enholm K (1999) where a new measure of software complexity called structural complexity is derived. (Torn et al., 1999). The Previous work focused on the derivation of all the parameters used in the model. However, this work will show that this model is applicable and useful to automatically computer software complexity. The aim of this research work is to validate Structural complexity model proposed in Torn *et al.* (1999) using the node representation.

The objectives of this research work are:

i. To prove that software complexity is independent of program length.
ii. To show that the complexity of a Software is dependent on the control Structure (construct) in a program.

2. LITERATURE REVIEW

In software engineering history, complexity measurement owns an importance. Kearney et al. (1986) Inappropriate use of software complexity measures can have large, damaging effects by rewarding poor programming practices and demoralizing good programmers. Software complexity measures must be critically evaluated to determine the ways in which they can best be used. In the using, people detect metric's shortcomings and issues. Therefore, people propose many methods to improve complexity metric. Even though, some issues of metric still can't be solved. So, a lot of metrics have been created. A number of useful related complexity measures projects have been reported in the literature. Neil (1994) cited a series of axioms and a number of measures that satisfy the axioms, the VINAP measures, to characterize the software complexity. Pearse and Oman (1995) applied a maintenance metrics index to measure the maintainability of C source code before and after maintenance activities. This technique allowed the project engineers to track the "health" of the code as it was being maintained. Maintainability is assessed but not in terms of risk assessment.

Schneidewind (2002) shifted the emphasis from design and code metrics to metrics that characterize the risk of making requirements changes. Although these software attributes can be difficult to deal with due to the fuzzy requirements from which they are derived, the advantage of have early indicators of future software problems outweighs this inconvenience. He developed an approach for identifying requirements change risk factors as predictors of reliability and maintainability problems. His case example consists of twenty-four Space Shuttle change requests, nineteen risk factors, and the associated failures and software metrics. Tran-Cao et al. (2005) used the Wood's task complexity model as a theoretical model to measure Software functional complexity which makes it possible to capture and quantify complexity. This model analyzes task, complexity in three dimensions: component complexity, coordinate complexity and dynamic complexity. The first two dimensions of the model to analyze software complexity at any early phase of the software life cycle (analysis phase). The third dimension proved to be difficult to capture at this stage and so was ignored in the model.

Woodward et al. (1979). This paper discusses the need for measures of complexity and unstructuredness of programs. A simple language independent concept is put forward as a measure of control flow complexity in program text and is then developed for use as a measure of unstructuredness. The proposed metric is compared with other metrics, the most notable of which is the cyclomatic complexity measure. Some experience with automatic tools for obtaining these metrics is reported. Olabiyisi (2005) design a machine that could calculates Haistead's volume, program level, program difficulty and cyclomatic number measures for different sorting algorithms to evaluate the complexity of these programs written in different implementation languages to investigate if such measures are applicable to algorithm comparison .It was discovered that the results obtained by using the machine designed agreed with those obtained when mathematical approach is used for the studied algorithms.

Landman et al. (2015). The study conducted an extensive literature study of the CC/SLOC correlation results and tested correlation on large Java (17.6 M methods) and C (6.3 M functions) corpora. The results show that linear correlation between SLOC and CC is only moderate as a result of increasingly high variance. It was further observed that aggregating CC and SLOC as well as performing a power transform improves the correlation. Revina et al. (2021). Here, the body of knowledge on complexity research and practice addressing its high fragmentation was further studied.. In particular, a comprehensive literature analysis of complexity research was conducted to capture different types of complexity in organizations. The results were comparatively analyzed, and a *morphological box* containing three aspects and ten features was developed. In addition, an established *multi-dimensional complexity framework* was employed to synthesize the results. Using the findings from these analyses and adopting the *Goal Question Metric*, a *method for complexity management was proposed*. This method serves to provide key insights and decision support in the form of extensive guidelines for addressing complexity. The findings can assist organizations in their complexity management initiatives.

Khoshgoftaar et al. (1994). In this paper, the results of a study conducted on a large commercial software system written in assembly language was reported. Unlike studies of the past, the data represent the unit test, integration, and all categories of the maintenance phase: adaptive, perfective, and corrective. The results confirm that faults and change activity are related to software measurements. In addition, the relationship between the number of design change requests and software measurements was also reported. This new observation has the potential to aid the software engineering management process. Finally, the value of multiple regression models over simple regression models was demonstrated.

3. METHODOLOGY

The research topic, the model and the sorting algorithms used in this research work were carefully chosen by the researcher. The unique feature of the model used in this work is the fact that no other complexity model found in literature has such a two-dimensional structure in representing the complexity. Therefore, the effect of such a model in calculating software complexity is of great interest to the researcher and therefore the one of the motivations for the research work. Sorting algorithms are used due to obvious reasons. In Computer Science, Sorting (of data) is of immense importance and is one of the most extensively researched subjects. It is one of the most fundamental algorithmic problems. So much so that it is also fundamental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc.

The Model

It is not wrong to say that there is a relationship between complexity and the length of the program, but all authors agree that when measuring complexity, one should take into account something different from length and length at the same time. This approach was followed in Törn et al. (1999) where a new measure of software complexity called structural complexity (effort-length Complexity metrics model) is derived. Below we present an overview of the software complexity model proposed in Törn et al. (1999), and then, we apply this methodology on some sorting algorithm in order to empirically validate it. An effort-length Complexity metrics model for software (ELC) based on measurement theory is introduced. The model is defined by:

$$e(p) = l(p)c(p)$$

Where p is any piece of software, e(p) is the effort in developing it, l(p) its length, and c(p) is the average structural complexity. For a collection of software units P = {P1, P2, ..., Pn} we calculate c(p) as the average of the individual units complexity:

$$c(P) = c(p_1, p_2, \ldots, p_n) = \frac{\sum_{i=1}^{n} l(p_i)\, c(p_i)}{\sum_{i=1}^{n} l(p_i)}$$

The individual unit lengths and total complexities are additive. So if we add them we obtain the length of the calculation (1(P)) and, respectively, the total complexity of the collection which in
this case is the effort (e(P)).

The classical technique to implement this principle is to restrict all control constructs to one of the three statements below:

i. Sequence
ii. Selection (decision)
iii. Iteration (loop)

In Törn et al. (1999) the authors use the above equations for the software collections and define new formulas/rules for computing structural complexity for the basic structural constructs discussed above that use some constants. The constants are different from one control structure to the other. Next we give the three formulas (sequence, Selection, iteration) for average
structural complexity using these constants.

Sequence

The sequence structure is the basic flow of a program; one statement follows another in sequence as they are coded. In sequence structure, there is no transfer of control to another point in the program. The entry point is the beginning of the sequence, the first statement; the exit point is the end of the sequence, the last statement. The formula therefore giving in the literature for average structural complexity for sequence is:

$c(p_1, p_2, \ldots, p_n) = c_s\ c(p_1, p_2, \ldots, p_n) =$

Let
i. q = {a;b}, where a and b are assignment statements. The node notation can then be written as:

(S(n 1 1)(n 1 1) = (n2 1.1) ⟷ l = 2; c=1.1 and e= 2.2

ii. p = {a; b; c}, where a, b, c are simple statements (assignment statements or defining statements). Then the equivalent node notation for the structure will be:

(s(n 1 1)(n 1 1)(n 1 1) = (n 3 1.1* (1+1+1)/3) = (n 3 1.1) ⟷ l=3, c = 1.1 and e = 3.3

Selection (choice)

In the selection structure, there is a condition for executing one or several instructions. If the condition is true, a set of distinct and separate statements will be executed. IF THEN ELSE, is a two-way selector common in a number of modern programming languages.

The formula therefore giving in the literature for selection is:

c (if)= $c_{if}\ c(b,p,q)$ - "if b then p else q"

Let

i. p = (If a then b else c), where a is a decision node and b, c are simple statements (program nodes). In node notation this will be written as:
(if (b 1 1) (n 1 1) (n 1 1)) = (n 3 1.3*(1+1+1)/3) = (n 3 1.3) ⟺ l=3, c = 1.3 and e = 3.9.

ii. p'= (If (a and b) then c else d), where a, b are decision nodes (Boolean expressions) and c, d are program nodes. Using node notation:

(if (b 2 1) (n 1 1) (n 1 1)) = (n 4 1.3) ⟺ l=4, c = 1.3 and e = 5.2.

iii. p" = (If a then (b and c) else d), where a is a decision node and b, c, d are program nodes. Using node notation:
(if (b 1 1) (s(n 1 1) (n 1 1))(n 1 1)) = (if (b 1 1) (n 2 1.1) (n 1 1)) = (n 4 1.3* (2+2.2)/4) = (n 4 1.365) l=4, e = 1.365 and e = 5.46.

Iteration (loop)
In an iteration structure, certain statements are executed repeatedly as long as a given condition is true.

The formula therefore giving in the literature for iteration is:

$c(while) = c_{do}\ c(b,p)$ " While b do p"

Let p' = (While a do b), where a is a decision node and b is a program node. In node notation this is written as:

(do (b 1 1) (n 1 1)) = (n 2 1.5). That is, the complexity density c=1.5 and the overall complexity e=3, and the length is 1 = 2.

Now let p' = (While a do (b and c)). In node notation, the structure will be:

(do (b 1 1) (s(n 1 1) (n 1 1))). This will be written further on as:
(do (b 1 1) (n 2 1.1)) = (n 3 1.5*(1+2.2)/3) = (n 3 1.6). This means that the average complexity (complexity density) is c=1.6, the length of the structure is 1=3, and the overall complexity is e=4.8.

In general, when applying the model, we consider c_s = 1.1 < c_{if} = 1.3 < c_{do} = 1.5 which is intuitive since we assign to the more complex structure a greater importance when calculating the complexity.
Using these formulas, the complexity of any program can be computed given the lengths and average complexities of the smallest parts (atoms): assignment statement, expressions, procedure calls and goto's. In our program, we consider all these to have the value of 1 (as it is suggested in Törn et al. (1999)).

In the sorting algorithms using C, Java and Visual BASIC programming language, consideration was not given to variable declarations when calculating the complexity. Also, when transforming "for" structure in "do" structure we have followed the rule:

for (i=0; i<n; i++) equivalent with (n 1 1) // i=0
(do (b 11)
….
(n 1 1) // i++)
)// close do

Programming languages Used
In this study, we apply this methodology on three sorting (Bubble, Heap and Shell) algorithms implemented in C, Pascal, Java and Visual BASIC adapted from Olabiyisi (2005) and try to validate it empirically.

Implementation
In this section the Structural complexity discussed above is implemented. The model is used on three sorting algorithms; and the results obtained compared. In order to compare different implementation languages, different implementation languages (C, Java, Pascal and Visual BASIC) for each of the sorting algorithms are used.

Implementing the Model with Sorting Algorithms.
For the work, the model described has been used to compute the complexity of the three of the sorting algorithms. To do so, the following actions were taken

(i) each of the three studied algorithms were coded using Pascal, C, Java and Visual BASIC resulting in four programs for each algorithm.
(ii) the same programming style (modular programming) was employed in the coding.
(iii) all the programs were run on the same computer.

For the work, we focused on the following.

i. Using the model, the structural complexity, program length and overall complexity were computed.
ii. Comparison of complexity of different sorting algorithms using the same implementation language.
iii. Comparison of complexities of different implementation languages for the same algorithm.

4. RESULTS AND DISCUSSION OF RESULTS

4.1 Results

Bubble Sort

Table 4.1a: C Language

void bubbleSort(int numbers[], int array_size)
{
int i, j, temp;
for (i = (array_size - 1); i >= 0; i--)
{
for (j=1;j<=i; j++)
{
if (numbers[j-1]> numbers[j])
{
temp = numbers[j-1]; numbers[j-1] = numbers[j]; numbers[j] = temp;
}
}
}
}
	(s(n 1 1)// i- (array_size - 1)
	(do(b 1 1)
	(s(n 1 1)//)-1
	(do(b. 1 1)
	(s(if(b 11)
	(s(n 1 1)
	(n11)
	(n 11)
	// close s
	close if
	(n11)
	//close s
	//close do
	(n11)
	// close s
	// close do
	//close s

The average structural complexity = 2.88; The program length 1-10; Overall complexity e-28.8

Table 4.1b: Java Language

public class Bubblesort
public void sort(int numbers[], int array_size){
int temp:
for (int i=(array_size-1); i>=0; i--){ for (int j = 1;
j<=i; j++){
if (numbers[j-1]> number [j]) {
temp = numbers[j-1];
numbers[j-1] = numbers[j];
numbers[j] = temp;
}
}
}
}
}
	(s(n 1 1)// int i=array_size -1
	(do(b 1 1)
	(s(n 1 1)// j = 1
	(do (b 1 1)
	(s(if(b 1 1)
	(s(n 1 1)
	(n. 1 1)
	(n. 1 1)
	)//close s
	)//close if
	(n11)
	)close s
	)// close do
	(n 1 1)
	) // close s
	) //close do
	)close s

The average structural complexity c =2.88; The program length l =10; Overall complexity e =28.8

Table 3.1b: Pascal Language

PROCEDURE BUBBLESORT (numbers: mynumbers).
VAR I, j: integer:
temp integer:
BEGIN
For i (array_size 1) DOWNTO 0 DO
BEGIN
J:=1:
while(j<=-i)DO
BEGIN
IF (numbers [j-1]>numbers[j]) THEN
BEGIN
temp:= numbers [j-1];
numbers [j-1]:-= numbers[j];
END;
j: =j+1;
END;
END;
END;
	(s(n 1 1)
	(n11)
	(n 11)
	(s(n 1 1)
	(do(b11)
	(s(n 11)
	(do(b11)
	(s(if (b 1 1)
	(s(n 1 1)
	(n11)
	)
	)
	)
	
	(n11)
	)//close do
	)
	)
	)
	)

The average structural complexity c = 2.43; The program length l = 11; Overall complexity e = 26.73

Table 4.1d: Visual BASIC

Public Sub Bubblesort(ByRef numbers() As Integer, ByVal array_size As Integer) Dim i As Integer Dim j As Integer Dim temp As Integer j=1 For i = array_size - 1 To 0 Step -1 Do Until (j > i) If (numbers(j-i)> numbers(j)) Then temp = numbers(j - i) numbers(j-i)= numbers(j) numbers(j) = temp j=j+1 Loop Next i End Sub	(s(n 1 1) (do(b 1 1) (do(b 1 1) (if(b 1 1) (s(n 1 1) (n 11) (n 11) (n 11)))) (n 1 1)))

The average structural complexity c = 2.68 ; The program length l=9; Overall complexity e =24.12

Heap Sort:

Table 4.2a: C Language

void heapSort(int numbers[], int array size)
int i, temp;
 	
for (i=(array_size/2)-1; i >= 0; i--) siftDown(numbers, i,
array_size);
 	
for (i= array_size-1; i >= 1; i--)
{
temp = numbers[0]; numbers[0] = numbers[i]; numbers[i] = temp;
sift Down(numbers, 0, i-1);
}
}
void siftDown(int numbers[], int root, int bottom)
{
int done, maxChild, temp;
done = 0;
while ((root*2 <= bottom) && (!done))
{
if (root*2 == bottom)
maxChild = root * 2;
else if (numbers[root * 2]> numbers[root * 2 + 1])
maxChild = root * 2;
else
maxChild = root * 2 + 1;
 	
if (numbers[root] numbers(maxChild
temp-numbers[root]:
numbers[root] numbers[maxChild). numbers[maxChild] = temp;
root maxChild;
else
done = 1;
}
}
	(s(n 1 1)//i=array size/2)-1
	(do(b 1 1)
	(s(n 1 1)
	(n 11)//i=array size-1
	(do(b 1 1)
	(s(n 1 1)
	(n 11)
	(n 11)
	(n 1 1)// i- -
	)//close s
	)//close do
	(n 1 1)// i—
	)//close s
	)// close do
	)//close s
	
	(s(n 1 1)
	(do(b 21)
	(if(b 1 1)
	(n 11)
	(if(b 1 1)
	(n 1 1)
	)
	(n 11)
	)// close if
	(if(b 1 1)
	(s(n 1 1)
	(n 11)
	(n 11)
	(n 1 1)
	)
	)//close if
	)//close s
	)//close do
	)//close s

Heap sort
The average structural complexity c = 2.36; The program length l = 11; Overall complexity e = 25.95

Sift Down
The average structural complexity c = 2.34; The program length l= 13; Overall complexity e = 30.42

Table 4.2b: Java

public class Heapsort{	(s(n 1 1)
public void sor(int number[], int array_size)[	(s(n 1 1)// i=aaray_size/2
int I, temp;	(do(b 1 1)
for(i= (array_ size/2; i>=0;i- -)	(s(n 1 1)
siftDown(numbers,I,array_size);	(n11) // i=array_size -1
for(i= (array_ size-1; i>=1; i- -) {	(do(b 1 1)
temp-numbers[0];	(s(n 1 1)
numbers[0]-numbers[i];	(n11)
numbers[i]=temp;	(n11)
sift Down(numbers, 0, i-1);	(n11)
}	(n11)// i--
}	)//close s
public void siftDown(int numbers[], int root, int bottom){	)// close do
int done, maxChild, temp;	(n 1 1)//i--
done =0;	)//close s
while((root*2 <= bottom) && (done != 0)){	)// close do
if (root*2 == bottom) maxChild = root*2;	(s(n 1 1)
else if(numbers [root*2] > numbers[root*2+1])	(n11)
maxChild = root*2;	(do(b 2 1)
else	(s(if (b 11)
maxChild = root*2+1;	(n 11)
if (numbers[root] < numbers [maxChild]){	(if(b 1 1)
temp = numbers[root];	(n 11)
numbers[root] = numbers(maxChild);	)// close if
numbers[maxChild] = temp;	(n 1 1)
root = maxChild	)// close if
}	(if(b 1 1)
else	(s(n 1 1)
done = 1;	(n 11)
}	(n 11)
}	(n11)
}	}//close s
	(n 11)
	)// close if
	)//close s
	)//close do
	)//close s
	)// close s

The average structural complexity c = 2.46
The program length 1 = 27
Overall complexity e = 66.42

Table 4.2c: Pascal

PROCEDURE heapsort (numbers: mynumbes),	(s(n+1)
VAR	(do(b 11)
i, temp:integer;	(s(n11)
BEGIN	(n 11)
i:- (array_size/2)-1;	)
WHILE (i>=0) DO	)
BEGIN	(n 11)
Sift Down (numbers, i, array_size);	(do (b11)
i:- i-1;	(s(n 1 1)
END;	(n 11)
	(n 11)
i:= array_size-1;	(n11)
WHILE (i>= 1) DO	)
BEGIN	)
temp:= numbers[0];	)
numbers[0]:= numbers[i];	
numbers[i]:= temp;	
siftdown(numbers, 0, i-1);	
END;	
END;	
PROCEDURE siftdown(numbers:mynumbers, rooty:integer, bottom: integer);	(s(n 1 1)
VAR	(s(do(b21)
done, maxchild,temp:integer;	(s(if(b 1 1)
done:=0;	(n 1 1)
BEGIN	(if(b 1 1)
WHILE (root*2<=bottom) AND NOT (done) DO	(n 11)
BEGIN	)
IF (root*2=bottom) THEN	(n 11)
maxchild:= root*2;	)
ELSE IF (numbers[root*2+1]) THEN	(if(b 1 1)
maxchild:=root *2;	(s(n 1 1)
ELSE	
maxchild root 2+	(n 11)
IF (numbers [root]	(n 11)
BEGIN	(n11)
temp numbers[root]	)
numbers[root] numbers(maxchild)	(n 11)
numbers[maxchild)- temp.	)
root: maxchild	)
END:	)
EISE	)
done:=1:	)
END:	
END:	

heapsort
The average structural complexity c = 1.64, The program length l=10; Overall complexity e = 16.4
Siftdown - The average structural complexity c = 2.34; The program length l= 14, Overall complexity e =32.76

Table 4.2d: Visual BASIC

Public Sub Heapsort(ByRef numbers() As Integer,	(s(n 1 1)
ByVal array_size As Integer)	(n11)
Dim i As Integer	(do(b 1 1)
Dim temp As Integer	(n 11)
j=1	(n 11)
For i = ((array_size/2) - 1) To 0 Step-1	)
Call siftdown(numbers, i, array_size)	(n11)
Next i	(do(b 1 1)
i = array_size - 1	(s(n 11)
Do Until i >= 1	(n11)
temp = numbers(i)	(n11)
numbers(0) = numbers(i)	(n 11)
numbers(i) = temp	(n11)
Call siftdown(numbers, 0, i - 1)	)
i=i-1	)
Loop	)
End Sub	
Sub siftDown(ByRef numbers() As Integer, ByVal root	(s(n 1 1)
As Integer, ByVal bottom As Integer)	(do(b 2 1)
Dim done As Integer	(if(b 11)
Dim maxchild As Integer	(n 1 1)
Dim temp As Integer	(if(b 1 1)
done = 0	(n 1 1)
While ((root* 2 < bottom) And Not (done))	)
If (root * 2 = bottom) Then	)
maxchild = root * 2	(if(b 1 1)
ElseIf (numbers(root * 2) > numbers(root * 2 + 1))	(s(n 1 1)
Then	(n 11)
maxchild = root *2+1	(n 11)
End If	(n 11)
If (numbers(root) < numbers(maxchild)) Then	)
temp = numbers(root)	(n 11)
numbers(root)= numbers(maxchild)	)
numbers(maxchild)-temp	)
root = maxchild	)
Else	
done = 1	
End If	
Loop	
End Sub	

heapsort
The average structural complexity c = 1.58
The program length l=12
Overall complexity e = 18.96

Siftdown
The average structural complexity e=2.16
The program length l= 13
Overall complexity e=28.08

Shell Sort

Table 4.3a: C Language

void shellSort(int numbers[], int array_size)	(s(n l1)
{	(do(b 11)
int i, j, increment, temp;	(s(n 1 1)// i=0
increment = 3;	(do(b 1 1)
while (increment > 0)	(s(n 1 1)
{	(n11)
for (i=0; i< array_size; i++)	(do(b 21)
{	(s(n 11)
j = i;	(n 11)
temp = numbers[i];	)
while ((>= increment) && (numbers [j-increment] >temp))	)
temp))	(n 11)
{	(n 1 1)// i++
numbers[j] = numbers [j - increment];	)// close s
j=j-increment;	)// close do
}	(if(b 11)
numbers[j] = temp;	(n 11)
}	(if(b 1 1)
if (increment/2!=0)	(n 11)
increment = increment/2;	)//close if
else if (increment == 1)	(n 11)
increment = 0;	)//close if
else	)//close s
increment = 1;	)// close do
}	)// close s

The average structural complexity c = 3.02
The program length l = 17
Overall complexity e = 51.34

Table 4.3b: Pascal

PROCEDURE Shellsort (numbers numarray, array size: int);
VAR
I, j, increment, temp:integer;
BEGIN
increment:-3:
WHILE (increment >0) DO
BEGIN
i:=0;
WHILE (i<array_size) DO
BEGIN
j:=i;
temp:=numbers[i];
WHILE ((>=increment) AND (numbers[j-increment>temp]);
BEGIN
numbers[j]:=numbers[j-increment];
j:=j+increment;
END;
numbers[j]:=temp;
i:=i+1;
END;
IF (increment/20 <>) THEN
increment:= increment/2;
ELSE
)
IF (increment :=1) THEN
increment:= 0;
ELSE
increment:=1;
END;
END;
	(s(n 1 1)
	(n 1 1)
	(s(n 1 1)
	(do(b 1 1)
	(s(n 1 1)
	(do(b 1 1)
	(s(b 11)
	(n11)
	(do(b 2 1)
	(s(n 1 1)
	(n11)
	)//close s
	)//close do
	(n 11)
	(n 1 1)
	)//close s
	) //close do
	(if(b 1 1)
	(n 11)
	(if(b 1 1)
	(n11)
	)
	(n 11)
	)//close if
	//close s
	)//close do
	)//close s
	)//close s

The average structural complexity c =3.09
The program length 1= 19
Overall complexity e = 58.71

Table 4.3c: Java

public class Shellsort{	(s(n 1 1)
public void sort(int numbers[], int array_size) throws Exception{	(do(b 1 1)
int i,j,increment,temp;	(s(n 1 1)// i=0
increment = 3;	(do(b 1 1)
try{	(s(n 1 1)
while (increment >0){	(n 11)
for (i=0; i < array_size;i++){	(do(b 21)
j=i;	(s(n 1 1)
temp = numbers[i];	(n 11)
while ((j>= increment) && (numbers[j-increment]>temp)){	)//close s
numbers[j]=numbers[j-increment];	)//close do
j-j-increment;	(n 11)
}	(n 1 1)// i++
numbers[j] = temp;	)//close s
}	)//close do
if (increment/2!=0)	(if(b 1 1)
increment = increment/2;	(n 11)
else if(increment == 1)	(If(b 1 1)
increment = 0;	(n 1 1)
else	)// close if
increment = 1;	(n 1 1)
}	)//close if
}	)//close s
catch(Exception e) {System.err.println(e);}	)//close do
}	(n 11)
}	)//close a

The average structural complexity e =2.92
The program length l = 18
Overall complexity e = 52.56

Table 4.3d: Visual BASIC

Public Sub ShellSort(ByRef numbers() As Integer, ByVal array_size As Integer)	(s(n 11)
Dim i As Integer	(do(b 1 1)
Dim i As Integer	n 11)
Dim increment As Integer	(do(b 1 1)
Dim temp As Integer	(s(n 1 1)
increment = 3	(n 1 1)
While (increment > 0)	(do(b 2 1)
For i = 0 To array_size - 1	(s(n 1 1)
j=i	(n 1 1)
temp = numbers(i)	)
While ((>= increment) And (numbers(j - increment)) > temp)	)
numbers(j) = numbers(j - increment)	(n 1 1)
j=j-increment	)
Loop	)
numbers(j) = temp	(1f(b 1 1)
Next	(n 1 1)
If (increment/2 <> 0) Then	(if(b 1 1)
increment = increment/ 2	(n 11)
ElseIf (increment = 1) Then	)
increment = 0	(n 11)
Else: increment = 1	)
End If	)
Loop	)
End Sub	

The average structural complexity c = 2.76
The program length l = 16
Overall complexity e = 44.16

4.2 Analysis of the Results: Complexities of the Sorting Algorithms Using The Model

Table 4.4a: Bubble Sort Complexity Measures by Different Implementation Languages.

S/NO	Algorithm Name	language	Average Structural Complexity	Program Length	Complexity
1.	Bubble	C	2.88	10	28.8
2.	Bubble	Java	2.88	10	28.8
3.	Bubble	Pascal	2.43	11	26.73
4.	Bubble	Visual Basic	2.68	9	24.12

Table 4.4b: Heap Sort Complexity Measures by Different Implementation Languages.

S/No	Algorithm Name	Language	Average Structural Complexity	Program Length	Complexity
1.	Heap	C	4.73	24	56.37
2.	Heap	Java	2.46	27	66.42
3.	Heap	Pascal	3.98	24	49.16
4.	Heap	Visual Basic	3.74	25	47.04

Table 4.4c: Shell Sort Complexity Measures By Different Implementation Languages.

S/no	Algorithm name	Language	Average structural complexity	Program length	Complexity
1.	Shell	C	3.02	17	51.34
2.	Shell	Java	2.92	18	52.56
3.	Shell	Pascal	3.09	19	58.71
4.	Shell	Visual Basic	2.76	16	44.16

Table 4.5: Complexity Measures of Different Sorting Algorithm in C Languages

S/no	Algorithm name	Language	Average structural complexity	Program length	Complexity
1.	Bubble	C	2.88	10	28.8
2.	Heap	C	4.7	24	56.37
3	Shell	C	3.02	17	51.34

Table 4.6: Complexity Measures of Different Sorting Algorithm in Java Languages

S/no	Algorithm name	Language	Average structural complexity	Program length	Complexity
1.	Bubble	JAVA	2.88	10	28.8
2.	Heap	JAVA	2.46	27	66.42
3	Shell	JAVA	2.92	18	52.56

Table 4.7: Complexity Measures of Different Sorting Algorithm on Pascal Languages

S/no	Algorithm name	Language	Average structural complexity	Program length	Complexity
1.	Bubble	Pascal	2.43	11	26.73
2.	Heap	Pascal	3.98	24	49.16
3	Shell	Pascal	3.09	19	58.71

Table 4.8: Complexity Measures of Different Sorting Algorithm In Visual BASIC Languages

S/no	Algorithm name	Language	Average structural complexity	Program length	Complexity
1.	Bubble	Visual BASIC	2.68	9	24.12
2.	Heap	Visual BASIC	3.74	25	47.04
3	Shell	Visual BASIC	2.76	16	44.16

Table 4.9: Program length of the three sorting algorithms implemented in the three programming languages

Algorithm name/ Language	Visual BASIC	Pascal	JAVA	C
Bubble	9	11	10	10
Heap	25	24	27	24
Shell	16	19	18	17

Table 4.10: Structural Complexity (Overall Complexity) of the three sorting algorithms implemented in the three programming languages

Algorithm name/ Language	Visual BASIC	Pascal	JAVA	C
Bubble	24.12	26.73	28.8	28.8
Heap	47.04	49.16	66.42	56.37
Shell	44.16	58.71	52.56	51.34

Table 4.11: Average Structural Complexity of the three sorting algorithms implemented in the three programming languages

Algorithm name/ Language	Visual BASIC	Pascal	JAVA	C
Bubble	2.68	2.43	2.88	2.88
Heap	3.74	3.98	2.46	4.7
Shell	2.76	3.09	2.92	3.02

4.2.2 Graphical Presentation of The Results

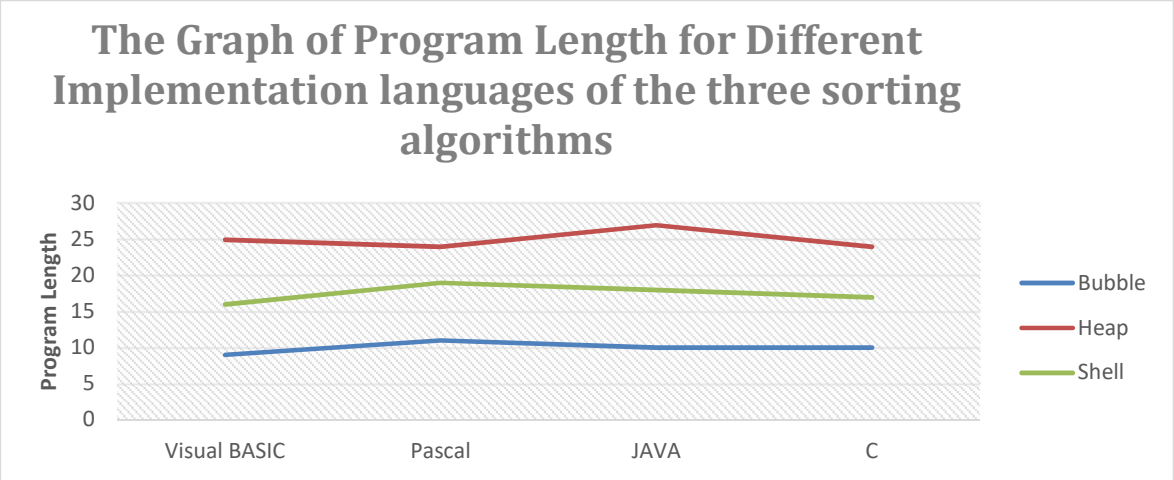

Figure 4.1: The Graph of Program Length for Different Implementation languages of the three sorting algorithms

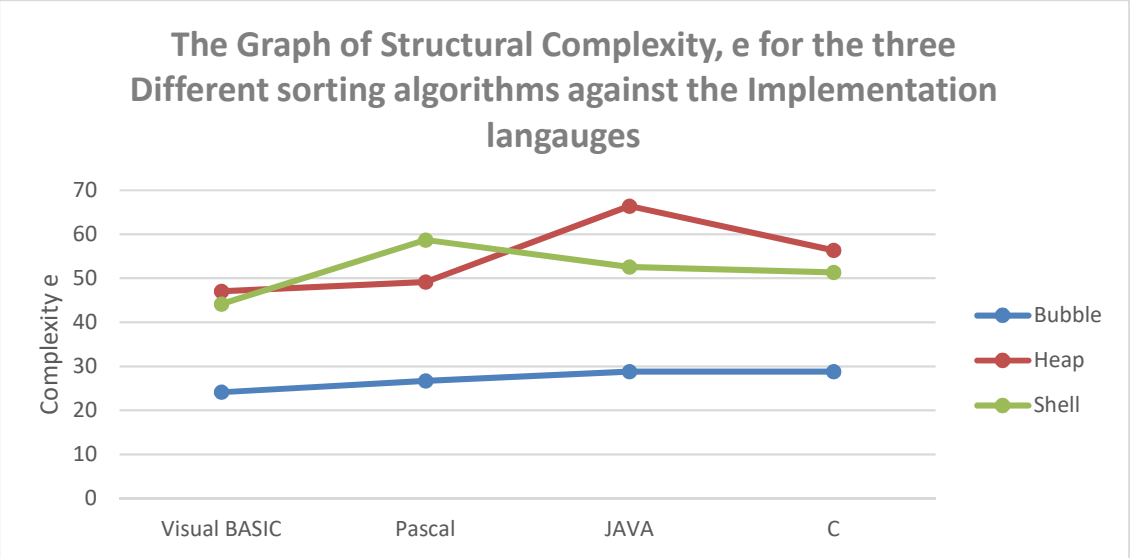

Figure 4.2: The Graph of Structural Complexity for the three Different sorting algorithms against the Implementation languages

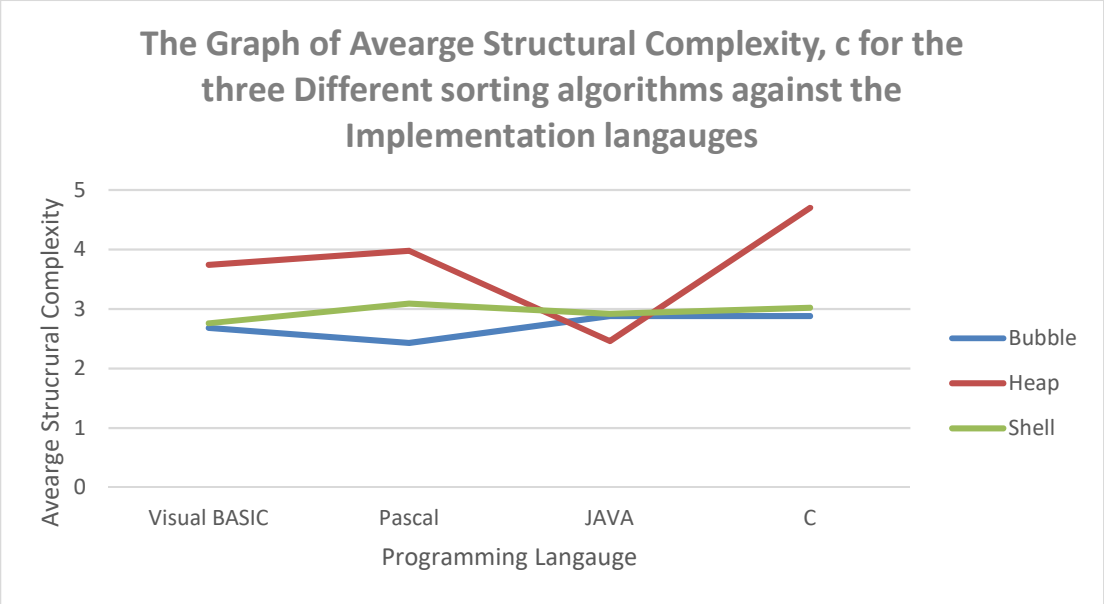

Figure 4.3: The Graph of Average Structural Complexity for the three Different sorting algorithms against the Implementation languages.

4.3 Discussion of Results

From the above figures and tables there are some interesting and important facts that could be observed. Figure 4.1, the Heap sort has the highest no of program length followed by shell while the bubble sort algorithm has the least program length. Figure 4.2 shows that heap sort has the highest structural complexity followed by bubble sort and shell sort. Bubble sort in all of the programming languages used has lower structural complexity. These agrees with the result obtained through mathematically analysis since heap sort has 0(n log n) complexities while bubble sort and shell sort have $0(n^2)$ complexity.

Figure 4.3 shows that Heap sort implemented in C language as the highest average structural complexity while Bubble sort implemented in Pascal has the lowest average structural complexity. All the sorting algorithms used are best implemented in Visual Basic. It can also be observed that Heap sort algorithm implemented in Java programming language has the highest structural complexity. Shell sorting algorithm implemented in Pascal programming language has the highest structural complexity in all of the sorting algorithms implemented in Pascal.

The Program length of shell sorting algorithms implemented in pascal is lower than that of heap sorting algorithm implemented also in Pascal but the structural complexity of shell sorting algorithm is higher than that of heap sort both implemented in Pascal language. This shows that taking just the length as a complexity measure is not a correct way to evaluate the quality of a software product of being complex or not. Even though the length is high, it is possible that the program is easily readable and easy to maintain, if it consists of few complex and nested structures and many simple statements.

5. SUMMARY, CONCLUSION AND RECOMMENDATIONS

With metric, the quality of software can be quantitatively described since you cannot control well what you cannot measure precisely. Software complexity is often used as index for readability, maintainability, testability and so on. In developing metrics, metrics can be classed as two kinds, macro and micro. Major macro complexity metrics are considered as the resource expended of software developing and the degree of difficulty. Micro metrics are based on program code, text-oriented or graph- oriented. This type of measure typically depends on program size, structure diagram and module interface. Moreover, micro complexity can be used to decompose program module and reduce system complexity. Precision is a big problem in measuring. Thus, new metrics are invented and famous metrics are improved. But these are not easy. Although there are many researches propose the metrics in finding software attributes, there are some deficiencies with these metrics. These basic observed deficiencies have been ignored in previous studies.

In this research work we have examined three sorting algorithm and used the structural complexity model by Torn et al (using node representation) to validate the model and results which are intuitively correct were obtained, that is we have obtained higher values for average structural complexity and total complexity for the programs which seems more complex than the others, not only in terms of lengths of program but also in terms of the contained structures.

The validity of the assumed complexity of each program construct (sequence selection, iteration) should be ascertained. Other issues for further research are to apply the complexity measures to another group of algorithm and also development of machine/tools to aid in computing program length, average structural complexity and the overall complexity rather than the manual method used in this research work for large systems. Since software now plays a very important role in our lives, we need to ensure that our software products are of good quality. The quality of software products can be seen as an indirect measure and is a weighted combination of different software attributes which can be directly measured.

It should also be noted that taking just length as a complexity measure is not a correct way to evaluate the quality of a software product of being complex or not. Even though the length is high, it is possible that the program is easily readable and easy to maintain, if it consist of few complex and nested structures and many simple statements, this was clearly shown in some of our programming examples. When measuring software complexity, we have to be very cautious on which metrics we use. The authors in Torn et al. (1999) state that one can obtain wrong result if we compare two different programs using one complexity measure and other two using another one. Another problem raised by the authors is that of establishing acceptable axioms for complexity measures. The failure to realize the existence of different views about complexity leads to conflicting axioms.

REFERENCES

1. Antinyan, V., Staron, M., & Sandberg, A. (2017). Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, *22*(6), 3057–3087. https://doi.org/10.1007/s10664-017-9508-2
2. Banker, R. D., Datar, S. M., & Zweig, D. (1989). Software complexity and maintainability. *Proceedings of the Tenth International Conference on Information Systems - ICIS '89*. https://doi.org/10.1145/75034.75056
3. COSTEA, A. (2007). On Measuring Software Complexity. *Journal of Applied Quantitative Method*, *2*(1). http://jaqm.ro/issues/volume-2,issue-1/pdfs/costea.pdf
4. GeeksforGeeks. (2017, November 7). *Software Engineering | Halstead's Software Metrics*. GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/
5. IJCSIS Editor. (2016). Journal Of Computer Science IJCSIS . In *Internet Archive*. IJCSIS PUBLICATION 2016 Pennsylvania, USA . https://archive.org/stream/JournalOfComputerScienceIJCSISFebruary2016/Journal%20of%20Computer%20Science%20IJCSIS%20February%202016_djvu.txt
6. Kearney, J. P., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (1986). Software complexity measurement. *Communications of the ACM*, *29*(11), 1044–1050. https://doi.org/10.1145/7538.7540
7. Khoshgoftaar, T. M., Szabo, R. M., & Woodcock, T. G. (1994). An empirical study of program quality during testing and maintenance. *Software Quality Journal*, *3*(3), 137–151. https://doi.org/10.1007/bf00402294
8. Landman, D., Serebrenik, A., Bouwers, E., & Vinju, J. J. (2015). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, *28*(7), 589–618. https://doi.org/10.1002/smr.1760
9. Neil, M. (1994). Measurement as an alternative to bureaucracy for the achievement of software quality. *Software Quality Journal*, *3*(2), 65–78. https://doi.org/10.1007/bf00213631
10. Revina, A., Aksu, Ü., & Meister, V. G. (2021). Method to Address Complexity in Organizations Based on a Comprehensive Overview. *Information*, *12*(10), 423. https://doi.org/10.3390/info12100423
11. Schneidewind, N. F. (2002). Investigation of the risk to software reliability and maintainability of requirements changes. *ACM Digital Library*. https://doi.org/10.1109/icsm.2001.972723
12. Team, G. L. (2020, December 12). *Time Complexity Algorithm | What is Time Complexity?* GreatLearning. https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/
13. Torn, A., Andersson, T., & Enholm, K. (1999). A complexity metrics model for software. *Uir.unisa.ac.za*, *24*. http://hdl.handle.net/10500/24371
14. Tran-Cao, D., Lévesque, G., & Meunier, J.-G. (2005). *Measuring software complexity for early estimation of development effort*. https://www.witpress.com/Secure/elibrary/papers/CMEM05/CMEM05003FU.pdf
15. Wikipedia Contributors. (2019, September 1). *Cyclomatic complexity*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Cyclomatic_complexity
16. Woodward, M. R., Hennell, M. A., & Hedley, D. (1979a). A Measure of Control Flow Complexity in Program Text. *IEEE Transactions on Software Engineering*, *SE-5*(1), 45–50. https://doi.org/10.1109/tse.1979.226497
17. Woodward, M. R., Hennell, M. A., & Hedley, D. (1979b, January). *IEEE Xplore - Temporarily Unavailable*. S3-Us-West-2.Amazonaws.com. https://ieeexplore.ieee.org/abstract/document/1702586